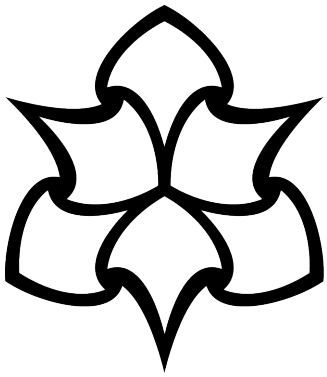

Manchester Metropolitan University

Lloyd, H ORCID logoORCID: https://orcid.org/0000-0001-6537-4036, Crossley, M ORCID logoORCID: https://orcid.org/0000-0001-5965-8147, Sinclair, M and Amos, M (2022) J-POP: Japanese Puzzles as Optimization Problems. IEEE Transactions on Games, 14 (3). pp. 391-402. ISSN 2475-1502

Downloaded from: https://e-space.mmu.ac.uk/628167/

Version: Accepted Version

Publisher: Institute of Electrical and Electronics Engineers

DOI: https://doi.org/10.1109/TG.2021.3081817

Please cite the published version

https://e-space.mmu.ac.uk

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

J-POP: Japanese Puzzles as Optimization Problems

Huw Lloyd, *Member, IEEE*, Matthew Crossley, Mark Sinclair, *Member, IEEE* and Martyn Amos

***Abstract*—Japanese puzzle games such as Sudoku and Futoshiki are familiar recreational pursuits, but they also present an interesting computational challenge. A number of algorithms exist for the automated solution of such puzzles, but, until now, these have not been compared in a unified way. Here we present an integrated framework for the study of combinatorial black-box optimisation, using Japanese puzzles as the test-bed. Importantly, our platform is extendable, allowing for the easy addition of both puzzles and solvers. We compare the performance of a number of optimization algorithms on five different puzzle games, and identify a subset of puzzle instances that could provide a challenging benchmark set for future algorithms.**

I. Introduction

JAPANESE culture has a long tradition of puzzles, dating back to the 1700s. Perhaps the most influential contemporary source of Japanese puzzles is the publisher Nikoli, who began producing a magazine in 1980, and who continue to this day [1]. The most famous "pencil puzzle" published by Nikoli is Sudoku, although other well-known puzzles include Kakuro, Futoshiki, and Slitherlink. Japanese puzzle games offer an interesting challenge for automated solvers, as they are generally computationally hard problems.

In this paper we present a *unified framework* for the solution of a number of Japanese puzzle games by several combinatorial optimization solvers (Ant Colony Optimization, Genetic Algorithms, Simulated Annealing, Backtracking, and Random Search). The core novelty of our approach lies in an abstraction architecture that allows solvers and puzzles to use a common interface. In this way, we may easily add both solvers and puzzles to the platform with minimal effort, as well as investigate the potential for "ensemble"-based solution methodologies. More fundamentally, our framework provides a consistent platform for the experimental comparison of algorithms for any puzzle that fits our interface, which allows us to easily perform experimental investigations into (a) the relative merits of different solution methods, and (b) the properties of different puzzles in terms of their difficulty. In this way, we both expand the available benchmarking repertoire (in terms of a codebase), and present Japanese puzzles as a relatively new and challenging set of problems with known properties.

The paper is structured as follows: in Section II we motivate the current work and provide some background. In Section III we describe our initial portfolio of five Japanese puzzles; for each puzzle, we give a brief introduction to its structure and rules, establish its complexity, and list known algorithms. In Section IV we describe our J-POP (Japanese Puzzles as Optimization Problems) platform, and in Section V we give the results of extensive experimental investigations, which both examine the relative difficulty of each of the puzzles, and assess the relative performance of each of our solvers. We discuss our findings in Section VI and conclude in Section VII by proposing several lines of future enquiry in this area.

Huw Lloyd and Matthew Crossley are with the Department of Computing and Mathematics, Manchester Metropolitan University, Manchester M1 5GD, UK.

Mark Sinclair and Martyn Amos are with the Department of Computer and Information Sciences, Northumbria University, Newcastle upon Tyne, NE1 8ST, UK.

II. Motivation and background

Board games have lain at the heart of artificial intelligence since the field's inception [2]; some of the earliest AI programs were developed to play checkers (draughts) [3] and chess [4], and attention has recently focussed on Go [5]. In addition, the past few decades have seen a rapid growth in the popularity and sophistication of *computer*-based games, and these have offered many opportunities for developments in the areas of non-player character behaviour learning, procedural content generation, and believable agents, to name but a few [6]. Here, we focus on two specific aspects of the crossover between AI and games identified in [6] - search and planning, and games as AI benchmarks - in the context of a class of games that is familiar to experts and non-specialists alike.

Puzzle games present a rich set of challenges for AI-based solvers in a "human-friendly" context - many such puzzles are a regular staple of newspapers and magazines [7]. Japanese pencil puzzles (JPPs) are a sub-class of puzzle games that share a number of characteristics [8]; (1) they are culturally-independent and language-neutral (that is, they do not rely on cultural context or knowledge of specific languages, unlike crosswords, which require both), (2) they are single-player, (3) they have simple rules, (4) each puzzle instance has a single, unique solution, and (5) they may be solved using only deduction (that is, no guessing is required).

Because of their accessibility and logical properties, JPPs have been used by mathematicians as a vehicle to teach principles such as counting, combinatorics, graph theory, and proof [9]. However, these properties also mean that they present a useful challenge to combinatorial optimization algorithms, as many JPPs are known to be NP-complete (for a fundamental reference, see [10], for a compendium of such puzzles, see [11], and for specific examples of NP-completeness proofs, see [12], [13], [14]).

Here, we present JPPs as the core of a new platform to facilitate investigations into combinatorial black-box optimization algorithms. Importantly, this platform uses a common abstract representation scheme for problems, meaning that new solvers and puzzles may easily be added. In line with previous recommendations for the generation of problem suites [15], our problems are derived from constraint-satisfaction problems; as we demonstrate, the platform facilitates the rapid analysis of

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

the properties of such problems as well as comparative analysis of the performance of a range of combinatorial optimization algorithms. Fundamentally, we offer a new practical platform for the rapid development of solvers, and the rapid application of existing solvers to new problems. Such a platform could find applications in theoretical investigations, the benchmarking of existing and new algorithms, and even as the basis for pedagogical work (for example, as the basis for competitions and student projects). In the next Section we describe the portfolio of puzzles provided in the current version of the platform.

III. Puzzle portfolio

In this Section, we provide short descriptions of each of the puzzles in our current portfolio, and provide references to a selection of existing solution methods.

A. Sudoku

We adapt our description of Sudoku from [16]. The simplest variant of Sudoku uses a 9×9 grid of cells divided into nine 3×3 subgrids (Figure 1 (left)). The aim of the puzzle is to fill the grid with digits such that each row, each column, and each 3×3 subgrid contains all of the digits 1…9 (Figure 1 (right)); any solution to a standard Sudoku puzzle is, therefore, a *Latin square* that meets the additional constraint that each 3×3 subgrid must contain the digits 1…9 [17]. An instance of Sudoku provides, at the outset, a partially-completed grid, but the difficulty of any grid derives more from the range of techniques required to solve it than the number of cell values that are provided for the player.

	6					5		2
	3				7			
	2		3		6			7
8	7	3		2	1	4	5	
9	4		5					
	1				4			
				9	5			4
3	9	4	8	1			7	5
	5	1		6	3		9	8

4	6	7	1	8	9	5	3	2
1	3	8	2	5	7	9	4	6
5	2	9	3	4	6	1	8	7
8	7	3	6	2	1	4	5	9
9	4	2	5	3	8	7	6	1
6	1	5	9	7	4	8	2	3
2	8	6	7	9	5	3	1	4
3	9	4	8	1	2	6	7	5
7	5	1	4	6	3	2	9	8

Fig. 1. Sudoku puzzle instance (left), and its solution (right).

Sudoku's NP-completeness was established in [10] (see [17] for a general overview). Notable approaches to solving Sudoku include formal logic [18], constraint programming [19], [20], evolutionary algorithms [21], particle swarm optimisation [22], [23], simulated annealing [24], tabu search [25], and entropy minimization [26]. Lloyd & Amos [16] describe an algorithm for Sudoku, based on Ant Colony Optimisation [27], and compare its performance to an iterated local search algorithm with constraint programming [28] and the deterministic algorithms of Knuth [29] and Norvig [30].

B. Futoshiki

Futoshiki is another Latin square-based puzzle, which takes place on an $n \times n$ board, in which every row and column must contain each digit $1 \ldots n$. Inequality constraints are specified by $>$ and $<$ signs placed between adjacent cells, such that one cell must be either greater than or less than its neighbour; the name Futoshiki means "inequality". An instance of the puzzle may be initially blank, or be partially-completed. An example Futoshiki puzzle is shown in Figure 2 (left), along with its solution (right).

This puzzle is NP-complete [31]; there exists relatively little work on its automated solution, although a candidate list strategy is given in [32] and approximation algorithms are described in [33].

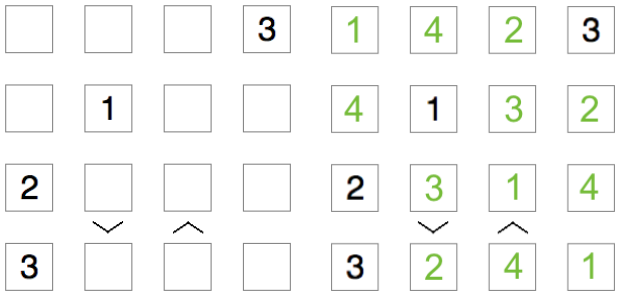

Fig. 2. Futoshiki puzzle instance (left), and its solution (right).

C. Hashiwokakero

Hashiwokakero (often abbreviated to "Hashi") is played on a grid with no fixed dimensions; the aim is to connect numbered "islands" with "bridges", such that the following constraints are observed:

- Each bridge must form a straight line connection between two islands, and must be orthogonal (i.e., no diagonal connections are allowed).
- No bridge may cross an island or another bridge.
- Any pair of islands may be connected by, at most, two bridges.
- The number of bridges connected to an island must be the same as the number label of that island.
- Bridges must connect islands such that they form a single connected component.

In Figure 3, we show an example Hashi puzzle and its correct solution. The NP-completeness of Hashi was demonstrated in [14]; there are relatively few published algorithms for this problem, although [34] presents a branch-and-cut method that solves instances with up to 400 islands.

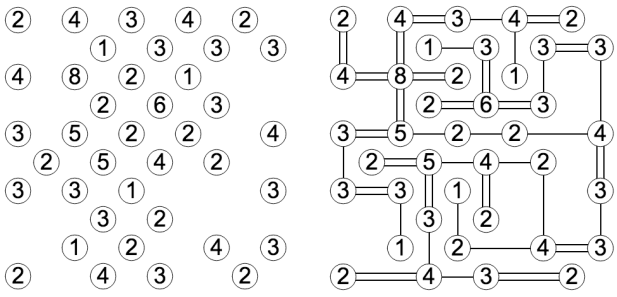

Fig. 3. Hashi puzzle instance (left), and its solution (right).

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

D. Nurikabe

We adapt our description of Nurikabe from [35]. The name of the puzzle is taken from that of a spirit in Japanese folklore, which manifests itself as an invisible barrier that impedes travellers. The connection derives from the basic aim of the puzzle, which is to construct a "wall" separating regions of the board. The puzzle is played on a rectangular grid of white cells, some of which initially contain numbers. A successful solution to the puzzle requires the player to shade in (colour black) non-numbered cells according to the following rules:

1) Black cells must form a single *continuous* region (the "wall").
2) Every numbered cell must occupy its own disjoint white region (an "island)" whose size is the same as the number label of that cell. The natural corollary of this rule is that islands may not touch, horizontally or vertically (immediate diagonal adjacency is allowed), as they would not be disjoint.
3) There must not exist any 2×2 black regions.

In Figure 4, we show an example Nurikabe puzzle and the correct solution, where each island contains a number of white squares that is equal to its labelled value, the black wall occupies a single continuous region, and no islands are touching. We also show, in Figure 5, an *invalid* solution, with broken constraints highlighted.

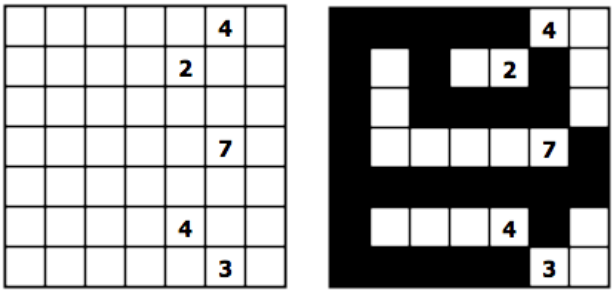

Fig. 4. Nurikabe puzzle instance (left), and its solution (right).

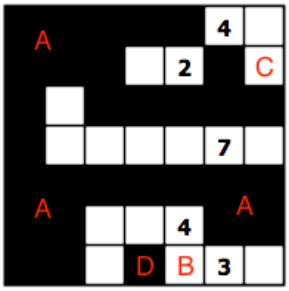

Fig. 5. Invalid Nurikabe solution, with various issues highlighted: (A) 2×2 blocks of black squares, (B) Island containing more than one value (which might be interpreted as touching "4" and "3" islands, (C) Island containing the wrong number of white squares, (D) Discontinuous wall.

The problem of solving Nurikabe is known to be NP-complete [36], [37], even under the restriction that islands may occupy no more than two cells. Early experimental work on solving Nurikabe used both Answer Set Programming and Constraint Programming [38], [39]. Another Nurikabe solution based on Constraint Programming [40] supplied the baseline for comparisons with a recent solution using Ant Colony Optimization [35].

E. Slitherlink

Slitherlink is played on a rectangular lattice of dots; some of the "squares" bounded by dots contain numbers. The objective of the game is to connect the dots using horizontal and vertical line segments, such that

- The complete line drawn forms a simple loop.
- Any numbered "square" must have the specified number of line segments immediately adjacent to it (i.e., if a square is numbered 0, then no line segments may "touch" that square).

Fig. 6. Slitherlink puzzle instance (left), and its solution (right).

An example Slitherlink instance is shown in Figure 6 (left), with its solution also shown. The problem of finding a solution to Slitherlink is NP-complete [10], [41], and a small number of algorithms for its solution have been proposed [42], [43].

IV. Platform Architecture

In order to compare the performance of a range of solvers on different puzzles types, we have developed an abstraction which comprises two components: the *simulator*, which is responsible for the iterative construction of candidate solutions to a puzzle under the direction of a *solver*, a process which makes choices from the sets of puzzle-agnostic options offered by the simulator. The solver aims to minimize the cost function reported by the simulator at the end of the solution construction process. In practice, these two abstractions are expressed as abstract base classes in an object-oriented language, with the various puzzles representing concrete implementations of the abstract simulator class, and the solvers concrete implementations of the abstract solver class. In this way, any solver may be easily connected to any simulator. In addition to the practical benefits of this architecture for implementing the solvers and puzzles, the platform also provides a method and framework for describing *black box* combinatorial optimization problems, in which solvers work only with abstracted solution components, interrogating the simulator for the cost value associated with particular combinations of these components but without any heuristic or domain-specific knowledge. We now describe the two abstractions, followed by the details of the concrete implementations for the various puzzles and solvers used in the experimental study. An overview of the architecture is given in Figure 7, and the implementations of the various solvers and simulators are outlined in subsections IV-C1 to IV-D5. Full details can be found in the reference code which may be downloaded from (https://github.com/huwlloyd-mmu/jpop).

A. Simulators

In our model, a *solution* S to a puzzle instance I is represented as an ordered set of n *solution components*

$$S = \{S_i, i \in [0, n-1]\}. \quad (1)$$

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

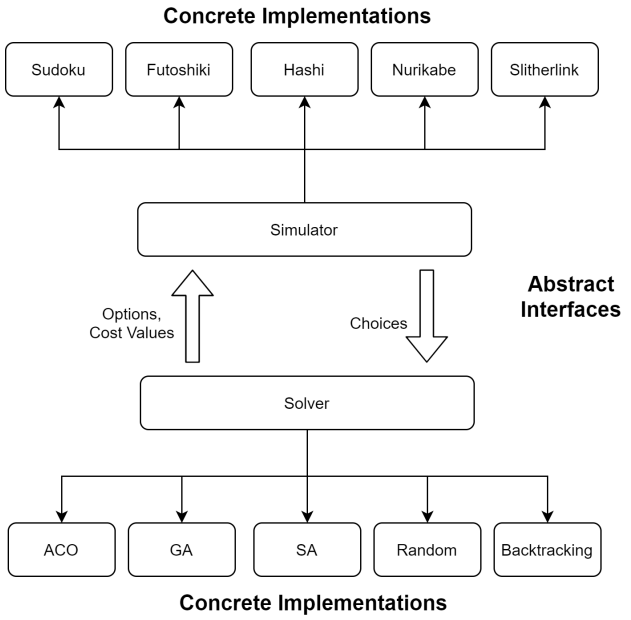

Fig. 7. The simulator-solver architecture, including the puzzles and solvers implemented in this work.

S is a subset of U_I, the set of all possible solution components for the puzzle instance. The solution components are uniquely-labelled elements of a puzzle solution, for example a value in a particular cell in Sudoku or Futoshiki, a particular cell coloured white in Nurikabe, or an edge connecting two specified vertices in Slitherlink or Hashiwokakero.

A simulator constructs a solution iteratively as follows. First, the simulator chooses some appropriate subset of U_I to offer to a solver as *options*. In practice these are encoded as integers which uniquely identify solution components, although it is important to note that a solver does not need to know how this is done, or even which puzzle the components refer to; they are treated as purely abstract entities. The solver makes a *choice* from the options, and this component is added to the the solution set S. The simulator then removes from U_I any solution components which are now forbidden by the *constraints* of the puzzle, before selecting another set of options for the solver. The process is repeated until U is empty. A cost function is then calculated, and returned to the solver.

Simulators may also introduce components whose function is to control some aspect of the construction process. For example, the Slitherlink simulator constructs a path on the grid, offering sets of edges as options, but requires a starting vertex. All possible starting points are given as the first set of options in a construction phase. These are labelled such as to differentiate them from the edges offered in later steps. The unique labelling enables solvers to learn good starting points, and to avoid starting the path construction at a vertex which is not part of the optimal solution.

Finally, it is important that the simulators offer options even in cases where there is only one valid possibility. This is so that the solvers see all the components in a solution, so that associations between components and cost values can be made, by whatever mechanism the solver uses (for example, the pheromone mechanism in Ant Colony Optimization).

B. Solvers

The solver is the controlling process which directs the construction of solutions by a simulator. Solvers may naturally work with iteratively constructed solutions (as in the case of Ant Colony Optimization or the random solver) or may use some internal solution representation which is used to control the simulator's construction process (as in the case of genetic algorithms and simulated annealing). In any case, the solver must employ some method for selecting from the options on offer from the simulator with reference to its internal solution representations, or any memory it carries of the association of particular components with good solutions. The procedure for constructing a solution is given in Algorithm 1.

Algorithm 1 Solution construction procedure

```
Reset simulator
do
    Get simulator options O ∈ U_I
    Choose component c ∈ O
    Set simulator choice c
while O ≠ {}
Get cost value from simulator
```

C. Simulator Implementations

1) Sudoku: The Sudoku simulator is based on the method described in [16]. Each cell in the grid maintains a set of possible values (1-9 in the case of a 9×9 instance, 1-16 for the 16×16 boards). The constraints are enforced each time the value in a cell is set according to the following rules:

1) The value is removed from the possible values of all cells in the same row, column and block (the *peer* cells).
2) If any peer cell now contains only a single possible value, that cell is also set.
3) If any peer cell is now the only cell in its row, column or block to contain a particular value, the peer cell is set to this value.

Note that since setting the value of a cell implies this propagation of constraints, this process is recursive. In cases where the puzzle is not solved, some cells will finish with no possibilities left. We take the number of these cells, which we call *failed* cells, in the final board as the cost function:

$$C_{\text{sudoku}} = N_{\text{fail}} \tag{2}$$

where N_{fail} is the number of failed cells. If $N_{\text{fail}} = 0$, then the puzzle is solved (by construction).

2) Futoshiki: The Futoshiki simulator employs a similar set of constraints to Sudoku, but peer cells (and constraints) are limited to only rows and columns; there are no blocks. However, there are also a number of inequality constraints between pairs of cells that must be maintained. Whenever the possible values of a cell are changed, or the value of a cell is set, all the inequality constraints of the cell, if any, are checked. If possible values of either cell would invalidate the constraint,

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

these are removed; if the constraint cannot be satisfied by the remaining values, both affected cells are failed, and all their remaining possibilities removed. If a cell is changed as a result of an inequality constraint, then all its inequality constraints must be checked recursively; however the recursion is only followed from greater to lesser ($>$) to avoid loops. In addition, should any cell be reduced to a single possibility, it will be set, and its peer constraints must also be recursively checked. As with Sudoku, when no choices remain, a cell is set to a *fail* state, and the cost function equals the number of failed cells:

$$C_{\text{futoshiki}} = N_{\text{fail}}. \tag{3}$$

If $N_{\text{fail}} = 0$, then the puzzle is solved.

3) Hashiwokakero: An instance of Hashiwokakero comprises a set of *nodes* which must be connected by edges in a single connected component. The graph is a multigraph, with a maximum of two edges connecting any pair of nodes. We define the components of the solution as the set of all possible edges between pairs of nodes which are horizontally or vertically adjacent; each node may be connected to up to four others, and each *edge site* may take values from 0,1,2 corresponding to no edge, one edge or two edges. In setting the value of an edge site, the constraints are imposed in three ways: firstly, when adding an edge, any crossing edge sites are set to the value 0 (indicating no edge), and the *edge capacities* of the two attached nodes (which are initialized to the value given by the puzzle instance) are decremented. Secondly, if the remaining capacity of the node is equal to the number of available edge sites, all edges are set. The final constraint, that the graph is connected, is captured in the cost function which is given by

$$C_{\text{hashi}} = N_{\text{comp}} - 1 + \sum_{v} |\deg(v) - N(v)| \tag{4}$$

where the sum is over the vertices v of the graph, $\deg(v)$ is the degree of vertex v, $N(v)$ is the puzzle's given value for node v, and N_{comp} is the number of connected components. A solution to the puzzle therefore has $C_{\text{hashi}} = 0$.

4) Nurikabe: The Nurikabe simulator uses the construction process described in [35]. The grid is initally filled with black cells, apart from the numbered cells. The islands are then visited in order, and 'grown' iteratively until either the island is full, that is it contains the number of cells given in the seed cell, or no further growth is possible given the constraints. The options offered by the simulator when growing an island are the adjoining cells which could be added to the island without breaking the constraints. The constraints imposed are:

- No cells which, if coloured white, would break the black cells into more than one connected component, are offered as options, and
- No cells which are adjacent to white cells from another island are offered as options.

The final constraint, that there can be no 2x2 block of black cells, is captured in the cost function, which is

$$C_{\text{nurikabe}} = N_{2\times 2} - N_{\text{white}} + \sum_{i \in \text{islands}} V_i \tag{5}$$

where $N_{2\times 2}$ is the number of 2×2 blocks of black cells, V_i is the given value for island i, and N_{white} is the number of cells coloured white. A solution to the puzzle will have $C_{\text{nurikabe}} = 0$.

5) Slitherlink: Slitherlink requires the solver to connect grid nodes with edges in a continuous loop, such that any numbered cell is bordered by the given number of edges. The slitherlink simulator operates in two phases. The first phase offers all nodes as potential starting points for the loop. In the second phase, the simulator constructs a walk around the board, at each step offering a set of possible undirected edges to add to the path. Once a node has been visited by the path, it is marked as unavailable, with the exception of the first node which is left open in order to allow the loop to be closed. The constraints imposed by the given numbers are imposed in two ways; firstly, no edge is offered which borders a cell containing the value zero; secondly, when an edge is added any values found in cells bordering the edge are decremented. In this way, once a cell is bordered by its correct number of edges, no new bordering edges may be added. The continuity of the path is guaranteed by construction and the constraint that it is a loop is enforced by a penalty in the cost function. This is given by

$$C_{\text{slitherlink}} = P_{\text{loop}} + \sum_{\text{cells}} V_{\text{cell}} \tag{6}$$

Where P_{loop} is a penalty imposed if the path is not a closed loop (we used $P_{\text{loop}} = 100$ in the experiments), and the sum is over the numbered cells, in which V_{cell} is the *remaining* value in the cell after solution construction (that is, the difference between the original value in the cell, and the number of edges bordering the cell). As with the previous puzzles, the optimum value of the cost function is zero.

D. Solver Implementations

1) Ant Colony Optimization: The Ant Colony Optimization (ACO) solver uses the *Ant Colony System* (ACS) variant of the algorithm, with the addition of the *Best-Value Evaporation* operator described in [16]. We include this addition since it was found to improve the performance of the algorithm in the case of Sudoku. The algorithm is the same as that described in [16] and [35] with the exception of the pheromone data structure, which is adapted to work with the simulator platform. We store pheromone values in an associative array which maps simulator *choices* onto floating point *pheromone* values. Let the set of keys in the associative array (corresponding to simulator choices) be K and the pheromone value associated with key k be $\tau(k)$, then we *read* the pheromone value associated with choice c using

$$\tau = \begin{cases} \tau(c), & c \in K \\ \tau_0, & \text{otherwise} \end{cases} \tag{7}$$

where τ_0 is the minimum pheromone value. The pheromone value τ is *written* to the pheromone data using the following procedure

$$K \leftarrow K + c \text{ if } c \notin K \tag{8}$$

$$\tau(c) = \tau. \tag{9}$$

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

The parameters for our ACS implementation are ρ_0, the minimum pheromone value, which we set to $1/N$, where N is the number of cells in the puzzle instance, q_0, the probability of making a 'greedy' choice in the random proportional rule, $\xi = 0.1$ (the standard ACS value), the local pheromone update parameter, ρ, the pheromone evaporation parameter, ρ_{BVE}, which controls best-value evaporation, and m, the number of ants. A full description can be found in [16] and [35].

2) Genetic Algorithm: We use a chromosome representation comprising an array of unsigned 16-bit integers. In order to convert a chromosome to a puzzle solution, the simulator is repeatedly queried for options, and one integer from the chromosome is consumed in order to make the selection. Let the options presented at a given step in the solution construction process be $O = O_1, O_2, \ldots, O_m$ and the next chromosome value be I. Then the choice made is O_c where $c = I \mod m$. The chromosome array is expanded as required; this is done by expanding the array and filling the new values with random numbers.

The genetic algorithm uses a fixed population P of N chromosomes $P_1, P_2, \ldots P_N$. At each generation, a new population is produced from the existing population as follows. With probability P_{xover}, crossover is used to generate two new population members. In this process, two parents are selected from the existing population using *tournament selection* based on the cost values of their associated solutions, and used to generate two new offspring using single point crossover. Otherwise (that is, with probability $1 - P_{\mathrm{xover}}$) a single population member is chosen, again using tournament selection, for propagation to the new generation. In all cases, the new population members are subject to *mutation* at a rate P_{mut}, in which integers from the chromosome are replaced with random values.

3) Simulated Annealing: Our simulated annealing implementation uses the same solution representation as the Genetic Algorithm. The local search operator replaces a single integer in the representation with a random value. Starting with an initial random solution, and a temperature $T = T_{\max}$, a candidate solution is produced at each step by applying the local search operator. Let the cost of the current solution be C_{cur} and the cost of the candidate be C_{new}. The probability, p_{accept}, of accepting the candidate solution is then given by

$$p_{\mathrm{accept}} = \begin{cases} 1 \text{ if } C_{\mathrm{new}} < C_{\mathrm{cur}} \\ \exp\left(\frac{C_{\mathrm{new}} - C_{\mathrm{cur}}}{T}\right) \text{ otherwise.} \end{cases} \tag{10}$$

The temperature is then updated after each step using

$$T \leftarrow T(1 - f_T), \quad f_T \in [0, 1] \tag{11}$$

4) Backtracking: A backtracking solver was implemented to allow comparison with a standard baseline algorithm for the solution of logic puzzles. The backtracking solver performs a depth-first search on options provided by the simulator. In order to do this, we introduce an ancillary data structure which maintains a stack of simulator states. The simulator state is pushed onto the stack when descending a level in the tree of simulator options, and popped when returning to a higher level. In practice, the pop operation works by replaying the stored options from the beginning to restore the state. The tree-traversal is achieved using recursion. When a dead-end is reached, i. e. the simulator returns an empty set of options, the evaluation count is incremented. The number of evaluations in the backtracking solver is therefore taken to be the number of dead-ends explored in the tree traversal. At each level of the search, options are visited in a random order. In this way, performing multiple runs on a single instance can explore the average performance of backtracking on a given instance.

5) Random: The random solver simply makes a random selection each time it is presented with a set of simulator choices during the process of solution construction, and keeps track of the best solution found so far.

V. Experimental evaluation

A. Solver Validation

In order to provide some confidence in our implementations of the solvers, we tested their performance on two standard problems. The first problem selected is the $N - k$ landscape with $k = 0$ [44], which validates the local search ability of the optimizers. The solvers should readily find the single optimum of this smooth, convex landscape, which we construct such that the probability of finding the optimum through chance is extremely low. The second validation problem is the uniform one-dimensional bin packing problem [45], which is an NP-complete problem with standard benchmark instances and well-known baseline heuristics against which the performance of the solvers can be compared. In the solver validation runs, we used a maximum of $200,000$ evaluations in each run, as in the pencil puzzle experiments which follow. The results show that the solvers perform as expected on these standard problems.

1) N-k Landscapes: We construct random, smooth ($k = 0$) $N - k$ landscapes with a known optimum by choosing a random binary vector V_{opt} of length M. The cost function for a vector V is then simply the Hamming distance $d(V, V_{\mathrm{opt}})$, and the cost value for the global optimum is 0. For a run of N evaluations, the probability of finding the optimum by chance is then

$$p = 1 - \left(1 - \left(\frac{1}{2}\right)^M\right)^N \tag{12}$$

We chose $M = 50$, which corresponds to a probability of finding a solution in $200,000$ evaluations of $p = 1.78 \times 10^{-10}$. Table I shows the results from 100 runs for solution rate (the fraction of runs in which the optimum is found), mean best cost value and the mean number of evaluations per solution for ACO, GA, SA and the random solver. All the stochastic algorithms perform very well, and rapidly find the solution in all runs. As expected, the random solver does not find the optimum in any run. The relatively high number of evaluations used by simulated annealing is due to the early (high temperature) part of the run making frequent uphill moves.

2) Bin-packing: An instance of the uniform, one-dimensional bin-packing problem comprises a set I of item sizes $s_i, i \in [1, N]$, and a bin capacity B. The problem is

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

TABLE I
RESULTS OF EVALUATING THE SOLVERS ON A SMOOTH N-K LANDSCAPE WITH A STRING LENGTH OF 50.

Algorithm	Success Rate	Mean Evaluations	Mean Cost
ACO	1.0	782.2	0.0
GA	1.0	1207.1	0.0
SA	1.0	48517.0	0.0
Random	0.0	-	9.6

TABLE II
MEAN VALUES OF K OVER 50 RUNS PER INSTANCE FOR THE UNIFORM ONE-DIMENSIONAL BIN-PACKING PROBLEM USING THE STOCHASTIC SOLVERS, COMPARED TO THE OPTIMUM AND FIRST-FIT HEURISTIC VALUES.

Instance Name	Optimum	First-fit	ACO	GA	SA	Random
u120_00	48	50	49.3	51.6	48.3	59.6
u120_01	49	51	49.4	50.8	49.0	59.2
u120_02	46	48	47.4	49.5	46.7	57.8
u120_03	49	52	49.2	50.3	49.0	58.6
u120_04	50	52	50.2	51.6	50.0	59.4
u120_05	48	52	48.3	49.9	48.0	58.3
u120_06	48	51	49.0	50.6	48.1	58.9
u120_07	49	52	50.8	53.0	50.2	61.1
u120_08	50	54	50.3	51.7	50.0	59.8
u120_09	46	49	47.2	49.1	46.5	57.5
u120_10	52	56	52.1	52.6	52.0	60.7
u120_11	49	52	49.4	50.2	49.0	58.7
u120_12	48	52	50.4	53.1	50.2	61.1
u120_13	49	51	49.1	50.3	49.0	58.3
u120_14	50	53	50.3	51.9	50.0	60.1
u120_15	48	53	48.1	49.1	48.0	57.5
u120_16	52	56	52.0	52.9	52.0	60.5
u120_17	52	56	52.6	53.9	52.0	61.8
u120_18	49	52	49.8	51.4	49.1	59.7
u120_19	49	52	49.1	50.0	49.0	58.4

to find a partition of I $(I_1, I_2 \dots I_K)$ which minimizes K subject to the condition that the sum of the item sizes in each partition is $\leq B$. We used the set of twenty instances from the file `binpack1` in the *OR-Library* dataset [46] which was first presented in [47]. All of these instances have a known optimum. We compared the performance of ACO, GA, SA and random solvers against the *first fit* heuristic. The solution representation is a permutation of the items, which is the order in which they are inserted into bins. Table II shows the mean value of K found over 50 runs for our solvers, the first-fit heuristic, and the optimum value in each case. We find that all the solvers perform better than random, and find averages that are close to the optimum and comparable to or better than the heuristic. Simulated Annealing is the best-performing algorithm, followed by ACO, and then the GA.

B. Puzzle Instance Data

We obtained puzzle instances from [48], which hosts a large collection of instances of a wide range of puzzles, complete with solutions. To the best of our knowledge, this is the most comprehensive source of puzzle instances currently available.

In order to select instances, we used an estimate of the size of the search space for each candidate instance, which is based on the probability of a random solver reaching the solution. For this we used a special solver (which we call the *prescient* solver) which 'solves' with full knowledge of the solution. At each step of solution construction, the fraction of the options presented to the prescient solver which are components of the known solution is determined. This is equal to the probability of a random solver making a correct decision at any given step in the construction process. A running product of these probabilities is maintained, and in this way the prescient solver is able to estimate the probability of arriving at the solution through random choices. We take the inverse of this probability as a measure of the size of the search space for an instance. Note that this measure should be treated with some caution, since in some of the puzzle types (such as Nurikabe) a solution component may be offered many times to the solver as part of sets of options of varying sizes. However, as confirmed by results in Section V, this number does seem to correlate well with the solution rate by a random solver.

For our initial dataset, we selected from [48] all instances of *Hashiwokakero*, *Futoshiki* and *Slitherlink*, all instances of *Nurikabe* without undetermined ('?') cells, and all square (9×9 and 16×16) instances of *Sudoku*. We then selected lower and upper limits to the search space size such that we eliminated any trivial instances (for which no searching is required), and produced at least 100 instances of each puzzle. For any puzzle with more than 100 instances in the range, we selected 100 at random from the available candidates. This set of 100 instances for each puzzle was further divided (randomly) into 10 training instances and 90 test instances. The training instances were used for tuning algorithm parameters (see Section V-C), while the test instances were used for the experimental evaluation. This exercise produced instances with search space sizes ranging from 12 to 1.12×10^{18}. Figure 8 shows scatter plots of the search space size against instance size (measured as the number of cells in the puzzle grid), split by puzzle type and membership of the train and test data sets for the pencil puzzle problems.

C. Parameter Tuning

In developing the solvers, we used typical values for the algorithm parameters. We retained these solver configurations for the experiments, but also used the *irace* package [49] to tune the algorithm parameters on the training set of instances. *irace* was run for each algorithm with a set of 50 training instances (ten each of the five puzzle types), and an experiment budget of 1500. We constructed a cost function which favours better solutions in cases where the puzzle is not solved, and which favours fewer evaluations to a solution in cases where the puzzle is solved (returning a cost value of 0). The cost function for tuning is:

$$C_{\text{tuning}} = \begin{cases} C \text{ if } C > 0 \\ \log_{10}\left(\frac{N_{\text{evals}}}{N_{\max}}\right) \text{ otherwise} \end{cases} \quad (13)$$

The best configurations found by *irace* are listed in Table III, along with the starting defaults.

D. Experimental Setup

We ran two configurations of SA, ACO and GA (default and tuned), along with the random and backtracking solvers 50 times on each the 450 puzzle instances in our test data

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

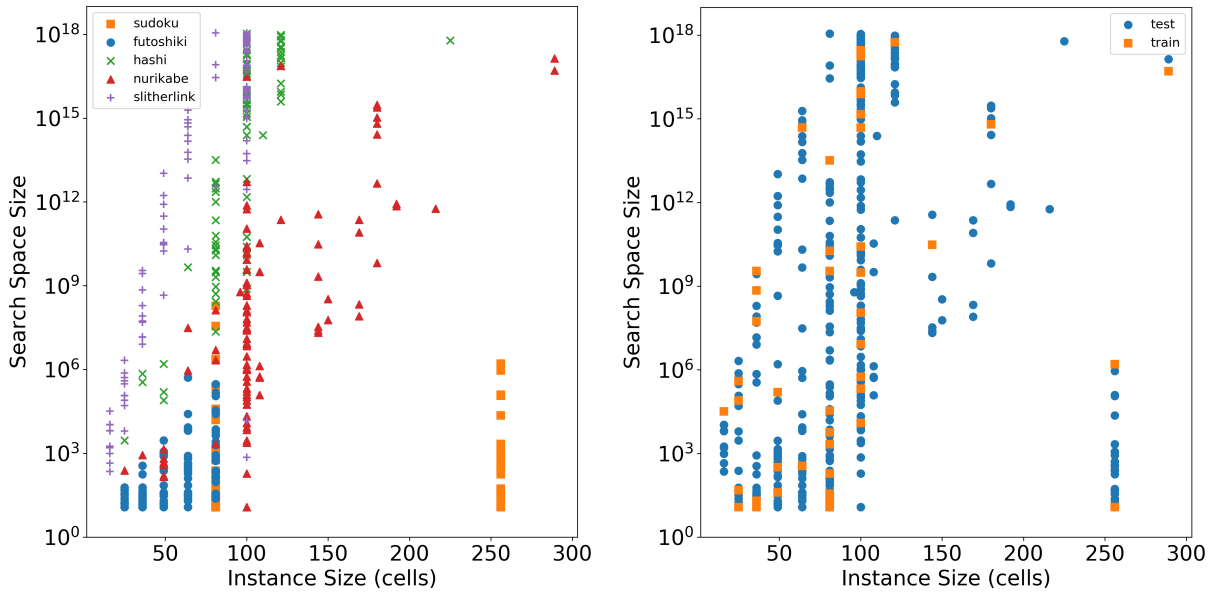

Fig. 8. Search space size, measured as the inverse of the probability of random solution, for the pencil puzzle instances used in the experiments. The data is shown split by instance type (left) and membership of the training or testing datasets (right).

TABLE III
DEFAULT AND TUNED PARAMETERS FOR THE GA, ACO AND SA SOLVERS.

Solver	Parameter	Default	Tuned
ACO	ρ	0.9	0.1623
	ρ_{BVE}	0.005	0.1697
	m	10	22
	q_0	0.9	0.4545
GA	N_{pop}	100	112
	N_{tourn}	10	37
	P_{xover}	0.98	0.1541
	P_{mut}	0.05	0.2866
SA	$T_{\max}$	1000	31.25
	f_T	0.000162	0.0693

set, using a maximum evaluation budget of 200,000 per run. For each run, we captured the best cost value found and the number of evaluations used when a solution was found. In the following sections we analyze the results to provide a number of different comparisons between the solvers and puzzles. For this analysis, we discarded the results from 14 instances which were found to have multiple solutions, which leaves 436 instances across all puzzle types.

E. Cost Value

Here, we analyze the behaviour of all the solvers in terms of the best cost value achieved at the end of a run. In this analysis, we treat the puzzles as optimization problems, regardless of whether the puzzle is solved or not, so a cost value of 0 represents a solution. We carried out two analyses which addressed the following questions:

1) For which instances do the solvers behave significantly better or worse than the random solver?
2) For which instances do the solvers behave significantly better or worse than all the other solvers?

We answer both these questions by performing statistical tests on the vectors of cost values produced by each solver in the 50 runs on each instance. We wish to detect statistically significant differences in the distributions of these cost values, for which the appropriate statistic is Mann-Whitney U, since the distributions cannot be assumed to be normal and the samples are not paired, and with the significance threshold modified as appropriate by the Bonferroni correction. For example, in testing the results of 2616 experiments (436 instances $\times 6$ solvers) to see if any show behaviour significantly differently to the random solver, we must divide our significance threshold (p-value) by the number of experiments. Similarly, in the tests for best or worst performance by a solver on a given instance, the significance threshold is divided by the number of solvers.

1) Comparison with Random Solver: Table IV gives the numbers of instances for which the performance was significantly better or worse than the random solver, for each of the six solvers. We use a significance threshold of 0.01, modified by the Bonferroni correction to give a p-value threshold of 3.82×10^{-6}. For a solver to show significantly different behaviour to random, a Mann-Whitney U test comparing the vectors of cost values produced by the given solver and the random solver on a particular instance must give a p-value less than this threshold. We see that all the solvers perform better than random in a large number of instances (typically ~ 200), but there are also a smaller number of instances for which the solvers perform significantly worse than random.

There are 29 instances for which at least one solver performs significantly worse than random. Of these, 22 are instances of Slitherlink, 6 are instances of Nurikabe, and one Hashiwokakero. Performance worse than random on this metric was not observed for any instances of Futoshiki or Sudoku. This suggests that Slitherlink and Nurikabe (and possibly Hashiwokakero) are capable of producing opimization prob-

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

TABLE IV
NUMBERS OF INSTANCES (FROM THE TOTAL OF 450) FOR WHICH SOLVERS ARE SIGNIFICANTLY BETTER OR WORSE THAN THE RANDOM SOLVER, AND ALL OTHER SOLVERS.

	Cost Value				Solution Rate			
	vs. Random		vs. All		vs. Random		vs. All	
Solver	Better	Worse	Best	Worst	Better	Worse	Best	Worst
ACO-1	193	27	5	44	154	36	5	27
ACO-2	215	6	146	4	223	16	146	3
GA-1	201	4	0	8	156	33	0	1
GA-2	178	0	7	0	178	10	7	0
SA-1	212	1	0	0	111	37	0	0
SA-2	200	2	0	3	112	46	0	0
Random	-	-	0	207	-	-	0	173

lems which at least some of our solvers find difficult. It is noteworthy that the two ACO algorithms seem to be more prone to this behaviour than the other solvers; ACO is strongly exploitative compared to the other solvers, and it is possible that some of these instances produce good local minima with few components in common with the global minimum, which can drive the ACO search in unproductive directions. More exploratory solvers such as GA and SA may be less prone to this issue.

2) Comparison Between Solvers: In this analysis, we look for solvers which performed better or worse than all the other solvers on a given instance, this time including the random solver. Again, we use the Mann-Whitney U test with a significance threshold of 0.01, modified using the Bonferroni correction by dividing by the number of solvers (7 – the six nature inspired optimizers and the random solver). Table IV gives the number of instances for which each solver is significantly best and worst amongst all solvers. The best performing solver is the tuned ACO configuration (ACO-2) which shows significantly better performance than all other solvers (on this metric) in 146 of the 436 instances. However, this configuration is also significantly worse than the others on 4 instances. The tuned genetic algorithm, GA-2, is the next best performing solver on this metric, and is not the worst solver on any instance. The random solver is by far the worst performing solver, which is perhaps to be expected, but it is nevertheless important to compare the solvers with random search to demonstrate that the search strategies give a significant improvement over chance. The strong performance of ACO over the other solvers may be due to the fact that the encoding used for ACO ties solution components to cost values more strongly than in the other solvers; for the encoding used in GA and SA, which only remembers the position of choices in the lists presented, a change (such as mutation) at one point in the chromosome will largely randomise the effect of the genes after that point. Conversely, this also makes the GA and SA solvers more exploratory, which could explain why these solvers are less prone than ACO to showing worse than random performance on some instances.

F. Solution Rate

For all puzzles we can compare the solution rates obtained by the solvers to each other, and to the solution rate found by the random solver. For comparing the success rates, we use Fisher's exact test, with a 2×2 contingency table comprising the number of successful and unsuccessful evaluations for each solver. The null hypothesis in this case is that the two solvers produce successes from evaluations at the same rate. We use the test to address the same two questions as in Section V-E.

1) Comparison with Random Solver: Table IV gives the number instances per solver for which the performance measured by the solution rate is significantly better or worse than the random solver, determined using the Fisher exact test with a significance threshold of 3.82×10^{-6} (0.01, after applying the Bonferroni correction).

The numbers are broadly similar to those for the cost value comparison, with the exception of the simulated annealing algorithms, which show far fewer instances with better performance. This is perhaps understandable since on many of the easier instances which are solved quickly by all solvers including random, the SA solvers will find a solution early in the annealing schedule when the solver is largely random. There are 58 unique instances for which at least one solver is worse than the random solver. These are spread across all puzzle types, with 15 Sudoku, 14 Futoshiki, 1 Hashiwokakero, 23 Slitherlink and 16 Nurikabe. The union of this set of instances and the 29 instances found in Section V-E contains 69 instances, which are enumerated by puzzle type in Table V. With the exception of Hashiwokakero, with only one instance, all puzzles are well represented; from our initial sample of 450 instances, we can in this way identify a sizeable subset of 69 for which at least one solver performed significantly worse than random. Taken together with the simulator-solver framework, this set could form the basis of a challenging benchmark suite for black box combinatorial optimization methods.

The success rates for each instance are compared to the probability of solution determined by the 'prescient' solver in Figure 9. Each instance for which at least one solution was found by a given solver is represented as a single point on the scatter plots, with the mean number of evaluations per solution on the y-axis, and the expectation value for the mean number of evaluations (which is equal to the search space size) on the x-axis. Points below the line $y = x$ represent better than random performance. All the solvers (with the exceptions of random and backtracking) show more points below the line than above, as may be expected, and the points for the random solver are closely clustered around the line. This confirms that the search space size determined by the prescient solver is a good guide to the expected performance of the random solver;

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

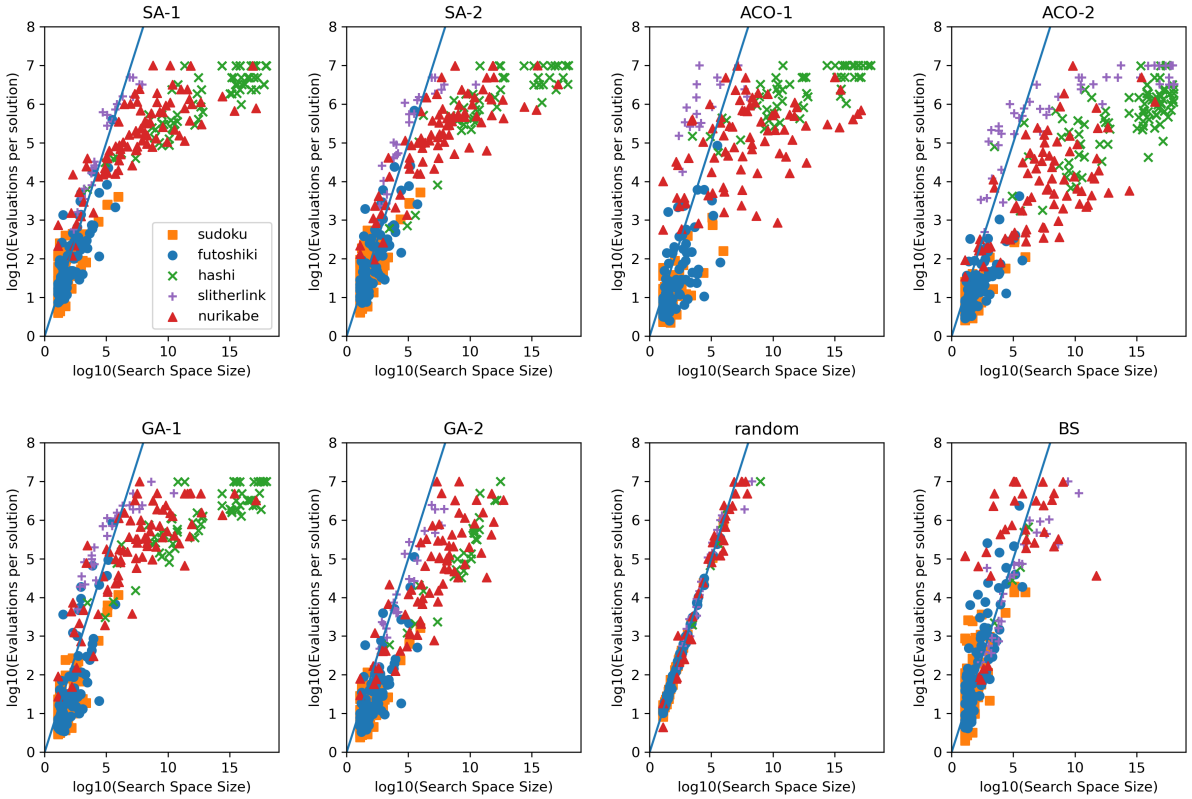

Fig. 9. Success rates for all instances compared to the probability of solution determined by the 'prescient' solver. The plots show the mean number of evaluations per solution compared to the expectation value (the search space size). For a given solver, only instances for which at least one solution was obtained are shown. Points below the solid line represent performance better than expected.

no solutions were found by the random solver for instances with search space sizes greater than 9.18×10^8. For 10^7 evaluations, we would expect solutions for instances with this search space size once in every 92 instances. The failure of the random solver to find a solution to an instance with a search space larger than this is therefore consistent with the hypothesis that our search space size measure is correct. All of the stochastic solvers are able to produce solutions to instances with search space sizes orders of magnitude larger than this. With the exception of the tuned genetic algorithm solver (GA-2), all solvers produce solutions across the full range of search space size, with search spaces up to 10^{18} being effectively searched for a single solution in 10^7 evaluations. In this case, the probability of finding a solution by chance is 10^{-11}. The fact that all the stochastic solvers tested here readily find solutions to these instances is a powerful demonstration of the ability of these algorithms to search efficiently in large combinatorial spaces.

G. Comparison with Backtracking Solver

For the comparison with the backtracking solver, we focus on the solution rate only. This is because the backtracking algorithm seeks to find the solution of a puzzle instance, and does not consider the cost value in any of the decisions made. Assessing the performance of backtracking as an *optimization* algorithm therefore makes little sense. In this comparison, we aim to answer the question of whether the stochastic optimization algorithms (ACO, GA, SA) perform significantly better or worse than the backtracking algorithm as *solvers*. The focus of this paper is on the treatment of JLP as *optimization problems*, however, it is important to validate the performance of the nature-inspired optimizers as *solvers* compared to a standard baseline solving algorithm for logic puzzles. Table VI shows the results, which are based on Fisher's exact test, as in the other solution rate experiments. Although all of the solvers perform worse than the backtracking solver on some instances, these are outweighed by a large factor by the instances for which the solvers outperform backtracking. The optimization algorithms, therefore, show strong performance as methods for the solution of logic puzzles, as well as in our derived optimization problems.

TABLE V
NUMBER OF INSTANCES OF EACH PUZZLE TYPE FOR WHICH ANY SOLVER BEHAVES SIGNIFICANTLY WORSE THAN RANDOM BASED ON COST VALUE, SOLUTION RATE, OR EITHER.

Puzzle	Cost	Rate	Union
Sudoku	0	15	15
Futoshiki	0	14	14
Hashiwokakero	1	1	1
Slitherlink	22	15	23
Nurikabe	6	13	16

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

TABLE VI
NUMBER OF INSTANCES FOR WHICH ANY OPTIMIZER SHOWS A SIGNIFICANTLY BETTER OR WORSE SOLUTION RATE THAN THE BACKTRACKING SOLVER.

Solver	Better	Worse
ACO-1	151	28
ACO-2	223	16
GA-1	146	23
GA-2	173	9
SA-1	111	24
SA-2	105	24

VI. DISCUSSION

Our experiments compared the performance of four stochastic combinatorial optimization algorithms (three nature-inspired algorithms, and random search) and a backtracking solver on a range of puzzle games implemented in a solver/simulator architecture which abstracts the details of the solvers and problems. We found that the three nature-inspired algorithms generally performed better than random search and backtracking, and in many cases the algorithms were able to find a single solution in a search space many orders of magnitude larger than the number of evaluations used in the search. The better-than-random performance of the algorithms is to be expected, nevertheless for a sizeable fraction of our problem instances (69 out of 450) at least one of the solvers performed *significantly* worse than random, either on the basis of the solution rate or the best cost value found. This suggests that the optimization problems we have produced from puzzle games are often difficult for standard optimization algorithms and therefore could form the basis of a benchmark set of problems with wider application. The proposed solver/simulator architecture enables new algorithms to be readily incorporated, and evaluated on the benchmark set.

The existence of instances for which some solvers perform worse than random search is perhaps to expected from the *No Free Lunch* (NFL) theorems [50]; the algorithms' ability to search some very large spaces efficiently is 'paid for' by poor performance on other instances. The NFL theorem implies that attempting to tune the algorithms for high performance on all instances would be futile, and a more productive approach may be the use of an ensemble of different solvers. Our results here show that all the solvers, including random, are significantly better or significantly worse than all the others on at least some instances. An ensemble of solvers, *including* random search, is therefore an appealing strategy for unseen instances or problems with unknown properties. The solver/simulator architecture facilitates this, and in future work will be exploited to bring the ensemble approach to bear on real-world optimization problems which, like the puzzles presented here, can be represented in the required form.

Many authors have studied the solution of puzzles using stochastic optimization algorithms (see Section I for details), often with the motivation of providing a new benchmark for, or insight into the behaviour of, a particular algorithm. However, in almost all cases, these studies treat a single puzzle and algorithm; a key contribution of this paper is to present, for the first time, a *unified* framework for the study of automated solution of *many* puzzles using a range of combinatorial optimization algorithms. The code for the framework is provided as open source, and may be freely used and extended in many other contexts.

VII. CONCLUSIONS

There are many avenues for future research which we intend to explore. From an experimental or practical standpoint, perhaps the most obvious would be the addition of more puzzles to the framework to extend the benchmark set, but a more fruitful direction may be the addition of more solvers (for example, learning automata, iterated local search, and tabu search) and problems derived from real-world contexts, such as the nurse-rostering problem. The current framework does not incorporate any domain-specific information, so the solution process of our solvers is different to that of human solvers. We might consider including problem-specific strategies in a future version of the platform; this could take the form of heuristic values passed to the solver along with the options. Some solvers, such as ACO, will readily make use of this information, whereas others may require some modification in order to exploit heuristics. Finally, a component-wise approach to the development of solvers may be an efficient method for the construction of new solvers with relatively little effort; this approach would depend on the creation of a range of pre-built solution representations and solver components that may be combined into new solvers using a domain-specific language.

From a theoretical standpoint, further study of the existing solver and instance set could be a fruitful area of research; it should be possible to instrument the search carried out by the solvers in a way which will shed light on both the behaviour of the solvers in searching the space, as well as the fitness landscape presented by the puzzles and instances. The unified representation of all solvers and puzzles at the interface of the solver/simulator architecture will allow them to to be directly compared using landscape analysis in terms of the abstract solution components. The practical extensions proposed in the preceding paragraph will only enhance the range of problems and solvers which can be investigated; in this way the framework can become a platform for the *theoretical* study of problems and algorithms, as well as their practical application.

AUTHOR CONTRIBUTIONS

Conceived and designed the study: (MA and HL). Puzzle simulators: Sudoku (HL); Nurikabe (HL and MA); Slitherlink (HL and MA); Futoshiki (MS); Hashiwokakero (HL). Solvers: ACO, backtracking, and random search (HL); GA and simulated annealing (MC). Performed experiments and analysed results: (HL). Drafted manuscript: (HL and MA). All authors contributed to the editing of the paper and approved the final manuscript.

REFERENCES

[1] A. Bellos, *Puzzle Ninja*. Guardian Books/Faber & Faber, 2017.
[2] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Springer, 2018, vol. 2.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2021.3081817

[3] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[4] C. E. Shannon, "Programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[5] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[6] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 317–335, 2014.

[7] M. Danesi, *The Puzzle Instinct: The Meaning of Puzzles in Human Life*. Indiana University Press, 2004.

[8] C. Browne, "Deductive search for logic puzzles," in *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[9] J. Rosenhouse and L. Taalman, *Taking Sudoku Seriously: The Math Behind the World's Most Popular Pencil Puzzle*. OUP USA, 2011.

[10] T. Yato and T. Seta, "Complexity and completeness of finding Another Solution and its application to puzzles," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 86, no. 5, pp. 1052–1060, 2003.

[11] G. Kendall, A. Parkes, and K. Spoerer, "A survey of NP-complete puzzles," *ICGA Journal*, vol. 31, no. 1, pp. 13–34, 2008.

[12] O. Ruepp and M. Holzer, "The computational complexity of the Kakuro puzzle, revisited," in *International Conference on Fun with Algorithms*. Springer, 2010, pp. 319–330.

[13] M. Holzer, A. Klein, M. Kutrib, and O. Ruepp, "Computational complexity of Nurikabe," *Fundamenta Informaticae*, vol. 110, no. 1-4, pp. 159–174, 2011.

[14] D. Andersson, "Hashiwokakero is NP-complete," *Information Processing Letters*, vol. 109, no. 19, pp. 1145–1146, 2009.

[15] O. M. Shir, C. Doerr, and T. Bäck, "Compiling a benchmarking test-suite for combinatorial black-box optimization: a position paper," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2018, pp. 1753–1760.

[16] H. Lloyd and M. Amos, "Solving Sudoku with ant colony optimization," *IEEE Transactions on Games*, vol. 12, no. 3, pp. 302–311, 2020.

[17] J.-P. Delahaye, "The science behind Sudoku," *Scientific American*, vol. 294, no. 6, pp. 80–87, 2006.

[18] T. Weber, "A SAT-based Sudoku solver," in *The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR): Short Paper Proceedings*, G. Sutcliffe and A. Voronkov, Eds., 2005, pp. 11–15.

[19] B. Crawford, M. Aranda, C. Castro, and E. Monfroy, "Using constraint programming to solve Sudoku puzzles," in *Third International Conference on Convergence and Hybrid Information Technology (ICCIT)*, vol. 2. IEEE, 2008, pp. 926–931.

[20] R. Lewis, "Metaheuristics can solve Sudoku puzzles," *Journal of Heuristics*, vol. 13, no. 4, pp. 387–401, 2007.

[21] C. Segura, S. I. V. Peña, S. B. Rionda, and A. H. Aguirre, "The importance of diversity in the application of evolutionary algorithms to the Sudoku problem," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 919–926.

[22] J. M. Hereford and H. Gerlach, "Integer-valued particle swarm optimization applied to Sudoku puzzles," in *IEEE Swarm Intelligence Symposium (SIS)*. IEEE, 2008, pp. 1–7.

[23] A. Moraglio and J. Togelius, "Geometric particle swarm optimization for the Sudoku puzzle," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 2007, pp. 118–125.

[24] Z. Karimi-Dehkordi, K. Zamanifar, A. Baraani-Dastjerdi, and N. Ghasem-Aghaee, "Sudoku using parallel simulated annealing," in *International Conference in Swarm Intelligence (ICSI)*. Springer, 2010, pp. 461–467.

[25] R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes, "A hybrid ac3-tabu search algorithm for solving Sudoku puzzles," *Expert Systems with Applications*, vol. 40, no. 15, pp. 5817–5821, 2013.

[26] J. Gunther and T. Moon, "Entropy minimization for solving Sudoku," *IEEE Transactions on Signal Processing*, vol. 60, no. 1, pp. 508–513, 2012.

[27] M. Dorigo and G. Di Caro, "Ant colony optimization: a new meta-heuristic," in *Proceedings of the 1999 Congress on Evolutionary Computation (CEC)*, vol. 2. IEEE, 1999, pp. 1470–1477.

[28] N. Musliu and F. Winter, "A hybrid approach for the Sudoku problem: using constraint programming in iterated local search," *IEEE Intelligent Systems*, vol. 32, no. 2, pp. 52–62, 2017.

[29] D. E. Knuth, "Dancing links," *arXiv preprint cs/0011047*, 2000.

[30] P. Norvig, "Solving every Sudoku puzzle," available at http://norvig.com/sudoku.html, accessed March 13, 2018.

[31] K. Haraguchi and H. Ono, "Approximability of Latin square completion-type puzzles," in *International Conference on Fun with Algorithms*. Springer, 2014, pp. 218–229.

[32] K. Haraguchi, "The number of inequality signs in the design of Futoshiki puzzle," *Journal of Information Processing*, vol. 21, no. 1, pp. 26–32, 2013.

[33] K. Haraguchi and H. Ono, "How simple algorithms can solve Latin square completion-type puzzles approximately," *Journal of Information Processing*, vol. 23, no. 3, pp. 276–283, 2015.

[34] L. C. Coelho, G. Laporte, A. Lindbeck, and T. Vidal, "Benchmark instances and branch-and-cut algorithm for the Hashiwokakero puzzle," *arXiv preprint arXiv:1905.00973*, 2019.

[35] M. Amos, M. Crossley, and H. Lloyd, "Solving Nurikabe with ant colony optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019, pp. 129–130.

[36] M. Holzer, A. Klein, and M. Kutrib, "On the NP-completeness of the Nurikabe pencil puzzle and variants thereof," in *Proceedings of the 3rd International Conference on FUN with Algorithms*, 2004, pp. 77–89.

[37] B. P. McPhail, "The complexity of puzzles: NP-completeness results for Nurikabe and Minesweeper," *Senior Thesis, Reed College*, 2003.

[38] M. Caylı, A. G. Karatop, A. E. Kavlak, H. Kaynar, F. Türe, and E. Erdem, "Solving challenging grid puzzles with answer set programming," 2007, available at http://research.sabanciuniv.edu/5086/1/puzzles-final.pdf.

[39] M. Celik, H. Erdogan, F. Tahaoglu, T. Uras, and E. Erdem, "Comparing ASP and CP on four grid puzzles." in *Proceedings of the 16th International RCRA workshop (RCRA 2009): Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, Reggio Emilia, Italy*, 2009.

[40] N. Tamura, "Nurikabe solver in Copris," available at http://bach.istc.kobe-u.ac.jp/copris/puzzles/nurikabe/.

[41] T. Yato, "On the NP-completeness of the Slither Link puzzle," *IPSJ SIGNotes ALgorithms*, vol. 74, pp. 25–32, 2000, (In Japanese).

[42] T.-Y. Liu, I.-C. Wu, and D.-J. Sun, "Solving the Slitherlink problem," in *2012 Conference on Technologies and Applications of Artificial Intelligence*. IEEE, 2012, pp. 284–289.

[43] R. Yoshinaka, T. Saitoh, J. Kawahara, K. Tsuruma, H. Iwashita, and S.-i. Minato, "Finding all solutions and instances of Numberlink and Slitherlink by ZDDs," *Algorithms*, vol. 5, no. 2, pp. 176–213, 2012.

[44] E. D. Weinberger, "Local properties of Kauffman's N-k model: A tunably rugged energy landscape," *Physical Review A*, vol. 44, no. 10, p. 6399, 1991.

[45] N. Karmarkar and R. M. Karp, "An efficient approximation scheme for the one-dimensional bin-packing problem," in *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 1982, pp. 312–320.

[46] J. E. Beasley, "OR library," available at http://people.brunel.ac.uk/ mastjjb/jeb/orlib/binpackinfo.html.

[47] E. Faulkener, "A hybrid grouping genetic algorithm for bin packing," *Journal of Heuristics*, vol. 2, pp. 5–30, 1996.

[48] A. Janko and O. Janko, "Raetsel, puzzles und anderer denksport," available at https://www.janko.at/Raetsel.

[49] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[50] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.